# 实验五：最短路径算法

**一、实验目的**

（1）理解路径松驰技术的思想

（2）掌握迪杰斯特拉算法（Dijkstra）的基本思想

**二、实验内容**

实现单源最短路经的迪杰斯特拉算法（Dijkstra），要求：分别采用数组、最小堆实现优先级队列。

1. **迪杰斯特拉算法（Dijkstra）基本思想**

1.设置一个顶点集合S，从源点s到集合中的顶点的最短路径的权值已确定

2.算法反复选择具有最短路径估计的顶点u∈V-S，并将u加入到S中

3.对u的所有出边进行松弛。

dijkstra算法的时间复杂度依赖于最小优先队列的具体实现。可以通过以下两种优先队列操作维护最小优先队列：

（1）采用数组实现：O(V^2+E)

（2）采用最小堆实现：O((V+E)lgV)

1. **实验过程**

**实验所用开发平台为vscode**

**实验代码描述语言为java**

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

import java.util.PriorityQueue;

import java.util.Queue;

import java.util.Scanner;

import java.util.Vector;

import javax.lang.model.util.ElementScanner6;

//public interface Comparable<T>

//此接口强行对实现它的每个类对象进行整体排序。

//这种排序被称为类的自然排序,类的 compareTo 方法被称为它的自然比较方法。

//实现此接口的对象列表（和数组）可以通过 Collections.sort（和 Arrays.sort）进行自动排序。

class Edge implements Comparable<Edge> {

int to; // 出边指向的元素

int cost; // 出边权重

// 定义构造方法

Edge(int to\_, int cost\_) {

this.to = to\_;

this.cost = cost\_;

}

// 实现Comparable接口唯一方法 : public int compareTo(Object obj);

// 该方法确定该类对象的排序方式

// 返回值对应待比较对象对应关系如下：

// 0 this == obj

// 负数 this < obj

// 正数 this > obj 前后交换

public int compareTo(Edge o) {

return this.cost - o.cost;// 升序排列

}

}

/\*\*

\* 数组实现优先级队列(升序)

\*/

class ArrPriorityQueue {

private int maxsize; // 队列最大长度

private Edge[] arr; // 队列对象数组

private int head; // head指示队头指针

public ArrPriorityQueue(int size) {

this.maxsize = size;

this.arr = new Edge[maxsize];

this.head = 0;

}

public void insert(Edge e) {

int j;

if (isEmpty()) {// 如果为空

arr[this.head++] = e;

} else if (isFull()) {

System.out.println("数组型优先级队列已满！");

} else {

for (j = this.head - 1; j >= 0; j--) {

if (e.cost > arr[j].cost) {

arr[j + 1] = arr[j];

} else {

break;

}

}

arr[j + 1] = e;

this.head++;

}

}

public Edge remove() {

if (isEmpty()) {

System.out.println("数组型优先级队列已空！");

return null;

} else {

return arr[--this.head];

}

}

public boolean isEmpty() {

return this.head == 0;

}

public boolean isFull() {

return this.head == this.maxsize;

}

public Edge getHead() {

if (isEmpty()) {

System.out.println("数组型优先级队列已空！");

return null;

} else {

return arr[head - 1];

}

}

public void displayPQ() {

if (isEmpty()) {

System.out.println("数组型优先级队列为空！");

} else {

for (int i = this.head - 1; i >= 0; i--) {

System.out.println(arr[i].toString());

}

}

}

}

/\*\*

\* 最小堆实现优先级队列

\*/

class heapPriorityQueue {

private int maxsize;

private Edge[] arr;// 基于堆的完全二叉树

private int head;

private int tail;

// 构造方法

public heapPriorityQueue(int size) {

this.maxsize = size;

this.arr = new Edge[maxsize + 1];

this.head = this.tail = 0;// 存储于arr[1..maxisze-1]中,arr[0]不使用,并作为优先级队列头指针

}

// 判空

public boolean isEmpty() {

return this.tail == 0;

}

// 判满

public boolean isFull() {

return this.tail == maxsize;

}

// 获取当前堆大小

public int getSize() {

return this.tail;

}

// 插入元素

public void insert(Edge e) {

if (isFull()) {

System.out.println("最小堆优先级队列已满");

} else {

arr[++tail] = e;

getFloat(tail);

}

}

// 删除并返回头元素

public Edge remove() {

if (isEmpty()) {

System.out.println("最小堆已空");

return null;

}

Edge min = arr[head + 1];

exchange(1, this.tail--);

getSink(head + 1);

return min;

}

// 交换数组元素

public void exchange(int index1, int index2) {

Edge temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

// 比较arr[index1]是否小于arr[index2]

public boolean less(int index1, int index2) {

if ((arr[index1].cost - arr[index2].cost) < 0) {

return true;

} else {

return false;

}

}

// 上浮操作

public void getFloat(int k) {

while (k > 1 && less(k, k / 2)) {

// arr[k] < arr[k/2],循环上浮

exchange(k, k / 2);

k = k / 2;

}

}

// 下沉操作

public void getSink(int index) {

while (2 \* index <= this.tail) {

int cindex = 2 \* index;

if ((cindex < this.tail) && less(cindex + 1, cindex))// 两个子数组元素选择一个更小的

{

cindex++;

}

if (!(less(cindex, index))) {

break;

}

exchange(index, cindex);

index = cindex;

}

}

}

public class Main {

static int maxn = 100; // 可输入的最大顶点个数

//static int[] l;

static int[] dis; // 存放从起点到各个顶点最短路径距离

static ArrayList<ArrayList<Edge>> e;// 每个顶点的出边集合

static ArrayList<ArrayList<Integer>> path; //从源点到每个终点的最短路径

/\*\*

\* 迪杰斯特拉算法

\* param1: 起点数组下标

\* param2: 优先级队列类型

\*/

public static void dijkstra(int s, int choose){

int vis[] = new int[maxn]; //值为1表示从源点s到该顶点的最短权值已经确定

dis = new int[maxn];

for (int i = 0; i < maxn; i++) {

dis[i] = 2147483647; //初始化起点到每个顶点的距离为无穷大

}

switch(choose)

{

case 0:

ArrPriorityQueue que = new ArrPriorityQueue(maxn);

que.insert(new Edge(s, 0));

dis[s] = 0;

//path.get(s).add(s);

while (que.isEmpty() == false) {

Edge now = que.remove();//算法选择当前队列中具有最短路径估计的顶点u

int u = now.to;

if (dis[u] < now.cost) {

continue;

}

if (vis[u] == 1) {

continue;//已经在vis[]中了，退出当前循环

}

vis[u] = 1;//将u加到vis[]中

for (int i = 0; i < e.get(u).size(); i++) {//对顶点u的所有出边进行松弛

int next = e.get(u).get(i).to;

int cost = e.get(u).get(i).cost;

if ((vis[next] == 0) && (dis[next] > dis[u] + cost)){

dis[next] = dis[u] + cost;

//最短路径中间过程更新操作

if (path.get(next).size() > 0) {

path.get(next).clear();

path.get(next).addAll(path.get(u));

path.get(next).add(next);

}

que.insert(new Edge(next, dis[next]));

}

}

}

break;

case 1:

heapPriorityQueue que2 = new heapPriorityQueue(maxn);

que2.insert(new Edge(s, 0));

dis[s] = 0;

while (que2.isEmpty() == false) {

Edge now = que2.remove();

int u = now.to;

if (dis[u] < now.cost) {

continue;

}

if (vis[u] == 1) {

continue;

}

vis[u] = 1;

for (int i = 0; i < e.get(u).size(); i++) {

int next = e.get(u).get(i).to;

int cost = e.get(u).get(i).cost;

if ((vis[next] == 0) && (dis[next] > dis[u] + cost)){

dis[next] = dis[u] + cost;

//最短路径中间过程更新操作

if (path.get(next).size() > 0) {

path.get(next).clear();

path.get(next).addAll(path.get(u));

path.get(next).add(next);

}

que2.insert(new Edge(next, dis[next]));

}

}

}

break;

default:

System.out.println("错误！输入的数字不符合要求！");

}

/\*\*

\* java自带的PriorityQueue(优先级队列),由于本实验是自己用数组和最小堆分别实现优先级队列,所以此处代码仅做测试和学习用

\*/

// Queue<Edge> que = new PriorityQueue<>(); //创建优先级队列

// que.add(new Edge(s, 0)); //起点到自己的权重为0,并加入到队列中

// dis[s] = 0;//起点到本身的距离为0

// while (que.isEmpty() == false) { //优先级队列若不为空,循环执行以下操作

// Edge now = que.poll(); //移除并返回队列头部的元素

// int u = now.to;

// if (dis[u] < now.cost) continue;//起点到顶点u的距离小于该有向边的权重,则当前直接结束本次循环,进行下一次循环

// if (vis[u] == 1) continue;//已经访问过顶点u,则直接结束本次循环,进行下一次循环

// vis[u] = 1;//标记访问了这个顶点u

// for (int i = 0; i < e.get(u).size(); i++) {

// //接下来开始访问顶点u,轮询u的所有出边,并加入到优先队列中

// int next = e.get(u).get(i).to;

// int cost = e.get(u).get(i).cost;

// if (vis[next] == 0 && dis[next] > dis[u] + cost) {

// //更新dis

// dis[next] = dis[u] + cost;

// que.add(new Edge(next, dis[next]));

// }

// }

// }

}

public static void main(String[] args) {

System.out.println("-----------------------实验5:迪杰斯特拉算法优化-----------------------");

System.out.println("请输入您采用的优先级队列类型,输入'0'为采用数组实现,输入'1'为最小堆实现");

Scanner scan = new Scanner(System.in);

int choose = scan.nextInt();

while (true) {

if (choose == 0) {

System.out.println("----------------您选择了用 '数组' 来实现迪杰斯特拉算法----------------");

System.out.println();

break;

} else if (choose == 1) {

System.out.println("---------------您选择用了 '最小堆' 来实现迪杰斯特拉算法---------------");

System.out.println();

break;

} else {

System.out.println("数字不符合要求,请重新输入。输入'0'为采用数组实现,输入'1'为最小堆实现");

choose = scan.nextInt();

}

}

e = new ArrayList<ArrayList<Edge>>();

path = new ArrayList<ArrayList<Integer>>();

for (int i = 0; i < maxn; i++) {

ArrayList<Edge> temp = new ArrayList<Edge>();

e.add(temp);

}

System.out.println("请您依次输入有向图的图中顶点个数、图中有向边条数、以及你选定的起点下标");

int n = scan.nextInt(); // 图中顶点个数

int m = scan.nextInt(); // 图中有向边条数

int s = scan.nextInt(); // 起点下标

//l = new int[maxn];

for (int j = 0; j < n; j++) {

ArrayList<Integer> item = new ArrayList<Integer>();

item.add(s);

path.add(item);

}

System.out.println("请您依次输入每条有向边的起始顶点、目标顶点、以及该有向边的权重大小");

for (int i = 0; i < m; i++) {

int from = scan.nextInt(); // 该有向边起点

int to = scan.nextInt(); // 该有向边终点

int cost = scan.nextInt(); // 该有向边权重

e.get(from).add(new Edge(to, cost)); // 加入到顶点的出边集合

}

dijkstra(s,choose);

for (int i = 0; i < n; i++) {

System.out.println("起点 " + s + " 到目的点 " + i + " 的最短距离为: " + dis[i]+", 路径为:"+path.get(i));

}

scan.close();

}

}

/\*\*

\* hjs的测试用例1：

\* 0

\* 5 10 0

\* 0 1 10

\* 0 3 5

\* 1 2 1

\* 1 3 2

\* 2 4 4

\* 3 1 3

\* 3 2 9

\* 3 4 2

\* 4 0 7

\* 4 2 6

\*

\* hjs的测试用例2：

\* 1

\* 7 10 0

\* 0 1 13

\* 0 2 8

\* 0 4 30

\* 0 6 32

\* 1 5 9

\* 1 6 7

\* 2 3 5

\* 3 4 6

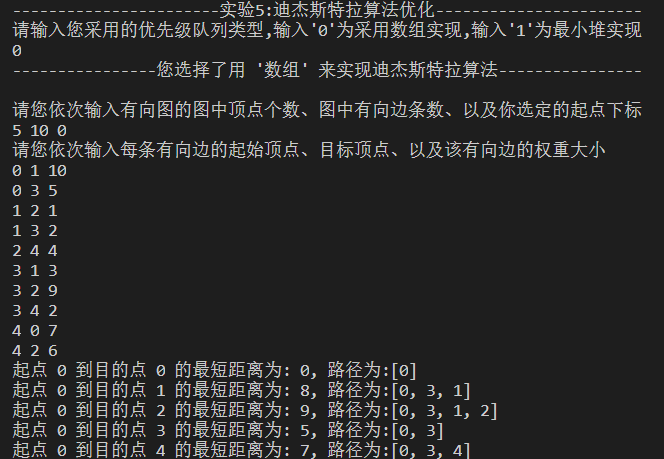
\* 4 5 2

\* 5 6 17

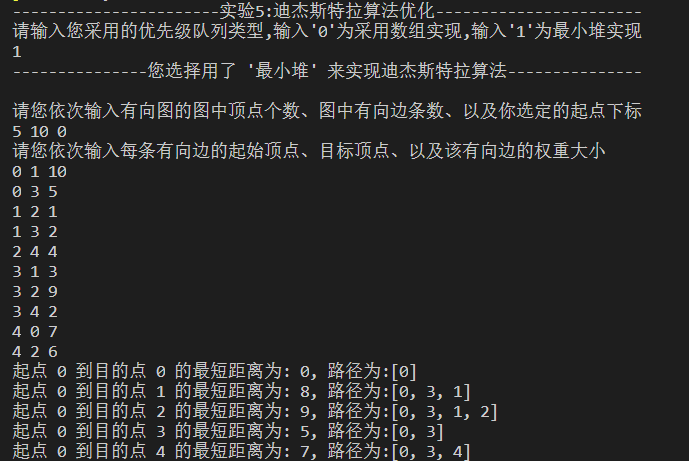
\*/

1. **实验结果**

测试样例1——数组实现：



测试样例1——最小堆实现：



测试样例2——最小堆实现：

